Improving Prediction Accuracy of Memory Interferences for Multicore Platforms

Cédric Courtaud, Julien Sopena, Gilles Muller, Daniel Gracia

▶ **To cite this version:**

Cédric Courtaud, Julien Sopena, Gilles Muller, Daniel Gracia. Improving Prediction Accuracy of Memory Interferences for Multicore Platforms. RTSS 2019 - 40th IEEE Real-Time Systems Symposium, Dec 2019, Hong-Kong, China. hal-02401625

HAL Id: hal-02401625
https://hal.inria.fr/hal-02401625

Submitted on 10 Dec 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Improving Prediction Accuracy of Memory Interferences for Multicore Platforms

Cédric Courtaud
Sorbonne Université/LIP6
Inria
Thales
cedric.courtaud@lip6.fr

Julien Sopena
Sorbonne Université/LIP6
Inria
julien.sopena@lip6.fr

Gilles Muller
Inria
Sorbonne Université/LIP6
gilles.muller@lip6.fr

Daniel Gracia Pérez
Thales
daniel-gracia.perez@thalesgroup.com

***Abstract*—Memory interferences may introduce important slowdowns in applications running on COTS multi-core processors. They are caused by concurrent accesses to shared hardware resources of the memory system. The induced delays are difficult to predict, making memory interferences a major obstacle to the adoption of COTS multi-core processors in real-time systems. In this article, we propose an experimental characterization of applications' memory consumption to determine their sensitivity to memory interferences. Thanks to a new set of microbenchmarks, we show the lack of precision of a purely quantitative characterization. To improve accuracy, we define new metrics quantifying qualitative aspects of memory consumption and implement a profiling tool using the VALGRIND framework. In addition, our profiling tool produces high resolution profiles allowing us to clearly distinguish the various phases in applications' behavior. Using our microbenchmarks and our new characterization, we train a state-of-the-art regressor. The validation on applications from the MIBENCH and the PARSEC suites indicates significant gain in prediction accuracy compared to a purely quantitative characterization.**

***Index Terms*—real-time, real-time systems, COTS, multi-core, interferences, memory interferences, characterization, profiling, experimental, provisioning, dynamic timing analysis, machine learning, inference**

I. INTRODUCTION

Commercial Off The Shelf (COTS) multi-core platforms offer computational power and energy efficiency at a low price, making them appealing targets for the development of complex embedded systems. Unfortunately, the adoption of COTS multi-core platforms is hindered by memory interferences which are due to the sharing of components of the memory hierarchy (caches, interconnects, DRAM chips and controllers,...) between cores, for cost and efficiency reasons. Memory interferences cause significant and hard-to-predict overheads, and they considerably challenge traditional timing analyses [1], which often results in unusably high estimated WCETs for real-time applications. In fact, accurately predicting memory interference overheads remains a difficult problem. Thus, the choice of a COTS hardware platform for a system requires a thorough study involving a substantial number of tests.

In this paper, we present a novel approach to estimate the interference overhead of an application based on a characterization of its behavior. Such estimation can be used in practice to quickly assess the suitability of a platform for a particular workload and ease the hardware provisioning of real-time systems. Our work is based on the observation that existing approaches to determine interference overhead rely only on bandwidth measurement which leads to conservative pessimistic values [2] [3]. We make three contributions. First, we introduce a new set of microbenchmarks that allow to cover a wide range of memory behavior varying both in nature and intensity. With these microbenchmarks, we show that the memory bandwidth leads indeed to compute inaccurate interference values. Second, we propose new metrics for quantifying the qualitative aspects of memory behavior. Since most of these metrics are not measurable using hardware counters, we have implemented a profiling tool using the VALGRIND[1] framework [4]. Our profiling tool generates high resolution profiles of the application memory behavior, allowing one to distinguish various phases in the execution of an application. Third, using random forest regressors [5], we compare how different characterizations of the memory traffic generated by our microbenchmarks perform for the inference of the overhead suffered by other applications. Results show a substantial gain of prediction accuracy with our new metrics.

Our results are as follows:

- We evaluate the effects of memory interferences on 1568 distinct cases of memory behavior on a iMX6.q Sabre Lite board [6]: a COTS platform, originally designed for automotive applications, which is currently widely used in the industry.
- We show the limits of a purely quantitative metric, with difference of execution time overhead exceeding 200% for similar observed bandwidth.
- We have implemented a profiler using the VALGRIND framework to generate high resolution profiles of the memory behavior. In a case study, we show that these profiles allow to split applications into phases of equivalent overhead.
- For different memory traffic characterizations, we evaluate how precisely random forests [5] trained on mi-

[1]The profiling tool and the microbenchmarks presented in this paper will be made available at the following url: http://julien.sopena.fr/ressources/memory-interference

crobenchmarks data can infer the overhead suffered by 78 phases from 29 applications of the MiBench and the PARSEC [7] suites. Compared to a purely quantitative characterization, our new metrics reduce both the average absolute and squared error of the validation set by respectively 50% and 74.4%.

This article is organized as follows. In Section II, we present our microbenchmarks and evaluate the range of behavior they cover. This section also presents our interference measure methodology. In Section III, we discuss of the quantitative characterization of memory consumption behaviors, define qualitative metrics, and presents our high resolution profiling approach. In Section IV, we evaluate the relevance of our new characterization. Finally, we present the related work in Section V, before concluding and discussing future work in Section VI.

II. Evaluating the impact of memory interferences

In this section, we present a methodology to study the impact of interferences on a given hardware platform. First we present a set of microbenchmarks that cover a wide range of memory behavior. Then we describe the experimental platform used in this article. This is followed by a description of our interference measurement protocol. Finally, we evaluate the range of sensitivity covered by our microbenchmarks.

A. Memory microbenchmarks

We consider an event based view of the memory behavior, as illustrated in Figure 1. The execution is represented by a sequence of instructions, and the memory traffic generated by an application is represented by a stream of shared memory system access requests emitted by a subset of instructions in its execution. We note N_{inst} and N_{access} the respective numbers of instructions and of memory accesses in an execution. We refer to the proportion of accesses per instruction as the *intensity* of the memory behavior, and to the number of accesses emitted by units of time as the *bandwidth*. The bandwidth is expressed in accesses per cycle in the equations. A memory access request is characterized by a *type* (read or write) and by the $address$ of the memory location it refers to.

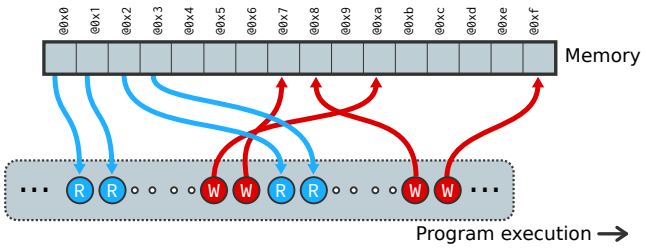


Fig. 1: Example of memory behavior

Our goal for the microbenchmarks is to generate a large number of different memory behaviors so as to evaluate interference overheads. There are existing microbenchmarks targeting memory such as STREAM [8]. However, most of them do not fit our needs because they are designed to evaluate the limits of memory system performance. Indeed, STREAM generates memory accesses as intensively as possible enforcing a unique access pattern.

We have designed a generic algorithm to vary memory behavior, whose pseudocode is given by Algorithm 1. It repeats an *access sequence* defined by five parameters allowing us to tune the intensity and the nature of the memory behaviour: a number of reads R and writes W, a throttle rate T, a read access policy P_R, and write access policy P_W. The access sequence follows a *fetch-compute-write back* pattern, each of its step being implemented by a corresponding loop. The fetch and the write back loops are in charge of generating memory traffic. The number of iterations of these loops is directly given by the R and W parameters. These two loops define the length of read and write bursts, and by extension the proportion and the interleaving of read and write access requests. The compute loop uses exclusively core private computation units. Its number of iterations is given by the throttle rate T multiplied by the number of fetch and write back loop iterations. Because it increases the number of instructions that do not produce memory accesses, T permits us to modulate the intensity of the memory behavior.

Algorithm 1 Microbenchmark structure

```
function STRESS((T, W, D, P_R, P_W))
    for i ← 1 to N do ▷ Repeat access sequence N times
        for j ← 1 to R do                          ▷ Fetch loop
            v ← v + S[P_R.next(i, j)]
        end for
        for j ← 1 to T(R + W) do                   ▷ Compute loop
            v ← local_computation(v)
        end for
        for j ← 1 to W do                          ▷ Write back loop
            S[P_W.next(i, j)] ← v
        end for
    end for
end function
```

The fetch and the write back loops are also governed by their respective access policies P_R and P_W. The access policy defines the type of data structures being walked and the sequence of addresses accessed according to the pattern being enforced. They are split in two groups summarized in Table I. The three policies in the first group implement a single array walk following different access patterns: sequential or random. The second group consists of policies redefining the one used to read data in the STREAM microbenchmark. In this policy, two values read from two distinct arrays are summed. The two arrays are walked sequentially and in parallel. We extend this policy, by varying the number of elements to be read, the type of data structures being traversed, and the access pattern enforced. Varying the number of elements read allows us to vary access interleaving. We also implement the sum of consecutive elements in a linked list because it involves a lot of data dependencies that may or may not be prefetched by

the target hardware. Finally, varying the access pattern allows us to vary the stress put on the prefetchers.

To build the benchmark suite, we retain thirteen of the 11^2 possible combinations of read and write access policies. Five are combinations of policies of the first group, two of these being particularly frequent in embedded systems. In the first case, data are read and written sequentially. Such behavior occurs for instance with the `memcpy` function. The second case corresponds to random reads followed by sequential writes. This behavior is found when data are gathered from various sources (sensors for instance). We also consider the duals of these behaviors, namely fully random accesses (random reads and random writes) and data scattering (sequential reads and random writes). Finally, we consider the case of lookup tables being used in the fetch and the write back loop, in order to mimic the case of the copy of linked data structures. The eight remaining combinations reproduce and extend the structure of STREAM: the read access policy is picked from the first group and data are written sequentially. To imitate the behavior of STREAM, we fixed the R and the W parameters. However the traffic can still be throttled.

B. Experimental platform

All experiments reported in this paper are conducted on the NXP iMX 6.q Sabre Lite board [6], [9]. The iMX6 processor targets among others the automotive market. The iMX6 processor is based on the Cortex A9 MPCore platform comprising four Cortex A9 cores. The Cortex A9 is a superscalar processor designed to offer good average performance, hence it relies on complex hardware features, notably caches, prefetchers and out-of-order execution.

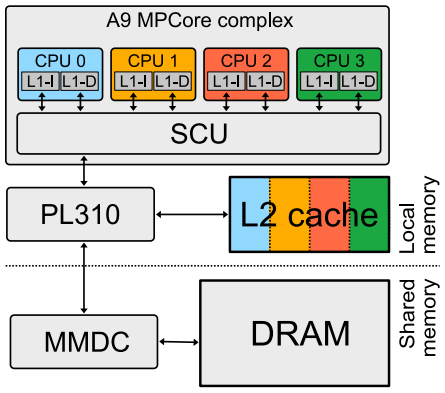


Fig. 2: iMX6 memory system block diagram

A simplified overview of the iMX6 memory system is depicted in Figure 2. A private 64KiB L1 harvard cache is associated to each core. The cores are connected to a *Snoop Control Unit (SCU)* in charge of maintaining L1 caches coherency and to managing the access to the L2 cache. The SCU is connected to a *PL310 cache controller* managing 1MiB of unified 16-way level 2 cache. The PL310 controller offers a *lockdown by master* [10] feature that allows one to set a mask for each core defining which way can be used by the cache eviction policy. We use this feature to split equally the L2 cache by allocating four disjoint ways to each core. Hence, it is not affected by spatial interference. However, as pointed by Valsan et al. [11], some contention can still occur on the SCU and the PL310 controler.

The last level of the memory hierarchy is the DRAM. Our platform features 1GiB DDR3 DRAM, with 8 banks of 128 MiB each. In our setup, this level is the only one which is not partitionned. The interface to the DRAM is the *Multi-Mode Memory Controller (MMDC)*, which is also in charge of the optimization of the global DDR bandwidth. To that end, it may perform access reordering and speculative row precharging [12], hence it can be unfair regarding access requests service time.

To comply with the event based model presented in section II-A, we decompose the memory system of our platform in a *private* and a *shared* part. We choose to include only memories subject to spatial interferences and their interface in the shared part. Consequently, since we partition the L2 cache, the shared part in the decomposition of our platform only consists in the DRAM and the MMDC. Thus, we consider that shared memory access requests are emitted on L2 cache misses.

The operating system used in this study is a GNU / LINUX distribution generated using the pyro release of the YOCTO project [13]. It uses the 4.1.15 kernel version compiled with GCC 6.4.0. Since our experiments do not involve scheduling and that applications are not preempted, we do not make use of `PREEMPT_RT` [14] patches or a platform like LITMUS RT [15].

C. Measuring interference

We determine the impact of interferences on an application by measuring the worst execution time overhead it suffers when the memory system is under contention. To this end we run the application *in isolation* (facing idle applications) and *in contention* (facing different combinations of *load* applications). Loads are a subset of our microbenchmarks selected to generate the greatest numbers of interferences. We compute the execution time overhead between the worst time measured in contention T_{cont} and in isolation T_{iso}.

$$Overhead = \frac{T_{cont} - T_{iso}}{T_{iso}} \quad (1)$$

We ensure that applications are not preempted and do not migrate. To that end, we pin each application on one core using the POSIX `sched_set_affinity` interface, and schedule them using the `SCHED_FIFO` policy with the maximum priority. To avoid kernel interferences, we disable *real time throttling* [16]. This mechanism is a safety net that periodically preempts applications scheduled with real time policies, in order to let the kernel manage software interrupts. This feature is enabled by default, and is a source of interferences in experimental results.

TABLE I: Implemented access policies

Name	Data structure	Access pattern	Description
sequential	one array	sequential	Simple array walk. Apply a fixed offset to the previous address
random	one array	random	Compute a random valid offset.
lookup	one array	random	Read sequentially the next entry of a shuffled array of offsets
sum-2	two arrays	sequential	Sum up two arrays sequentially. **Similar to STREAM.**
sum-3	three arrays	sequential	Sum up three arrays sequentially.
sum-2-r	two arrays	random	Sum up two arrays. Two random offsets are computed.
sum-3-r	three arrays	random	Sum up three arrays. Three random offsets are computed.
sum-2-l	linked list	sequential	Add two consecutive elements of linked lists. Nodes are contiguous in memory
sum-3-l	linked list	sequential	Add three consecutive elements of linked lists. Nodes are contiguous in memory
sum-2-lr	linked list	random	Add two consecutive elements of linked lists. Nodes are shuffled in memory
sum-3-lr	linked list	random	Add three consecutive elements of linked lists. Nodes are shuffled in memory

TABLE II: Pair of access policies used in the microbenchmarks

	R	W	D	P_R	P_W
linear	✓	✓	✓	sequential	sequential
scatter	✓	✓	✓	sequential	random
gather	✓	✓	✓	random	sequential
random	✓	✓	✓	random	random
lookup	✓	✓	✓	lookup	lookup
sum-2	×	×	✓	sum-2	sequential
sum-3	×	×	✓	sum-3	sequential
sum-2-r	×	×	✓	sum-2-r	sequential
sum-3-r	×	×	✓	sum-3-r	sequential
sum-3-l	×	×	✓	sum-2-l	sequential
sum-3-l	×	×	✓	sum-3-l	sequential
sum-3-lr	×	×	✓	sum-3-lr	sequential
sum-3-lr	×	×	✓	sum-3-lr	sequential

D. Impact of memory interferences on microbenchmark instances

By varying the parameters of Algorithm 1, we obtain 1568 microbenchmark instances with memory behavior of varying nature and intensity. The data set comprises instances of all the access policy combinations defined in Table II. The R and W parameters are determined by multiplying a read over write ratio (0, 0.25, 0.5, 0.75, and 1) with a total number of accesses (20 and 100). The range of the throttle parameter T varies from 0 to an upper bound that depends on the combination of access policy. The upper bound is 10,000 for the STREAM extensions, and 2000 for the other combinations. The reason of this difference is purely practical, as large throttle values results in longer experiments.

The relationship between the overhead and microbenchmarks' parameters is shown in Figure 3. For each nature of traffic observed (defined by all the benchmark parameters except the throttle rate), we can associate a curve representing the evolution of the overhead in function of the throttle rate. In Figure 3a, we can see that if each curve is decreasing exponentially with the throttle rate (the x scale is logarithmic), the speed of decay of each curve varies greatly. Figure 3b exhibits important variations observed for the same throttle values. The overhead varies between 109% and 384% for a throttle of 0, between 49% and 262% for a throttle of 10, and between 7% and 199% for a factor of 100. In Figure 3c, the two highlighted curves illustrate how the rate of decay may vary. There is a 178% overhead difference in favor of the red curve for a throttle of 0. They suffer roughly the same overhead for a throttle of 8, and for a throttle of 100 the difference is of 115% in favor of the nature illustrated by the blue curve. This shows a great variety of shapes between the various nature of memory consumption, in spite of the fact they share a fairly similar structure.

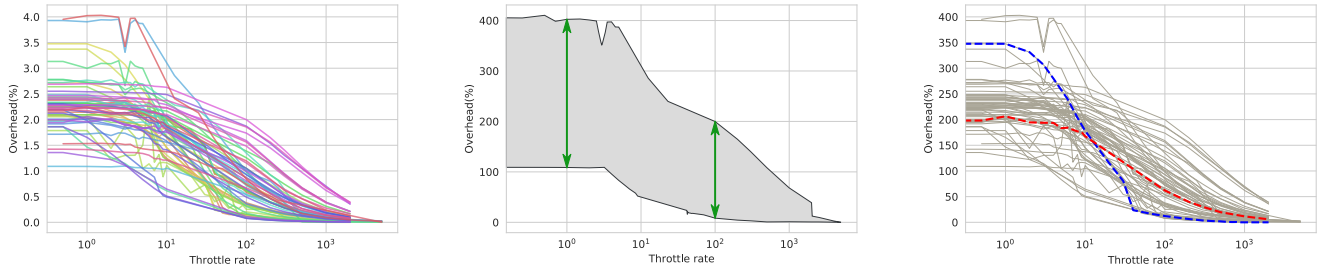
From this first result, we can make two conclusions:

1) Our microbenchmarks cover a wide range of sensitivity cases.
2) The intensity of memory consumption clearly plays a great role in the sensitivity to interferences, although it is not sufficient to explain it completely.

There is clearly a need for a better characterization of the nature of memory behaviour.

III. Characterizing memory usage patterns

To transpose our microbenchmark results to real applications, we must quantify relevant aspects of memory behavior regarding sensitivity to memory interferences. We can distinguish two types of metrics: *quantitative* metrics related to the intensity of the memory behavior and *qualitative* metrics related to its nature. While many quantitative aspects are easily measurable using hardware performance counters, this is not the case for most qualitative metrics. To avoid this shortcoming, we emulate a simplified version of the target architecture to capture the traffic yield explicitely by the application, and record it in a novel high resolution profile format. With this approach, we can quantify yet unmeasurable aspects of the memory behaviour. Additionally, an application



(a) (b) (c)

Fig. 3: Impact of memory interferences on microbenchmark instances.

profile allow to distinguish various phases in the application execution.

The rest of this section is organized as follows. In the first place, we study the quantitative characterization of memory behavior using bandwidth. In a second place, based on our previous results and our knowledge of the experimental platform we propose some qualitative metrics. In a third place, we present our profiling platform, and our high resolution profile. We conclude this section with a case study of an application from the PARSEC suite.

A. Quantitative characterization

We characterize the intensity of memory behavior with the bandwidth to shared memory, as it is fairly common, notably in regulation systems such as MemGuard [2] or the approach proposed by Blin et al [3]. We use the counters provided by the *Performance Monitoring Unit (PMU)* of the MMDC to measure it. Since the MMDC is shared between cores, the counters only account the global bandwidth and do not differentiate the consumption of individual cores. Hence, the bandwidth of an application can only be measured in isolation, although this is not a problem to characterize applications memory behavior.

Figure 4a depicts the relation between the overhead suffered by our microbenchmarks and their bandwidth *in isolation*. Just like on the figure 3, we can associate a curve to each nature of traffic which is defined by all the microbenchmark parameters except the throttle rate. Here we draw curves expressing a linear relationship between the overhead and the bandwidth in isolation. The speed at which the overhead grows with the bandwidth, the slope of these curves, varies greatly. Let's show that the slope of these curves is in fact the *average delay suffered per access*. The average delay suffered per access is the difference of the average access latency observed in contention and in isolation.

$$D_{access} = CPI_{mem}^{cont} - CPI_{mem}^{iso} \tag{2}$$

Since this is an average value, it does not vary from access to access. Hence, we can state that the overall delay suffered by an application is the average delay per access repeated N_{access} time.

$$T_{cont} - T_{iso} = N_{access} \cdot D_{access} \tag{3}$$

Injecting this results in equation 1, we show the relationship between the overhead an application suffer and its bandwidth in isolation.

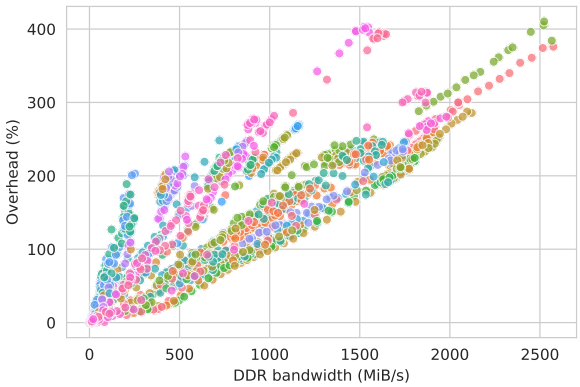
$$\begin{aligned} Overhead &= \frac{T_{iso} - T_{cont}}{T_{iso}} \\ &= \frac{N_{access}}{T_{iso}} \cdot D_{access} \\ &= BW_{iso} \cdot D_{access} \end{aligned} \tag{4}$$

Reasoning in terms of slopes allows us to quantify the variation of sensitivity of our microbenchmarks independently of their bandwidth. In figures 4b, we can see that the slope observed for each nature of traffic is mostly constant with two notable exceptions: important slopes (over 200 cycles) are unstable, and some nature of traffic exhibit a slight change of slope for high throttle values. The range of observed slope values is significant. If we apply the highest observed slope (481.51) to the highest observed bandwidth (0.084 access/-cycle), we have a 4,074% overhead. Fortunately, we can see in Figure 4c that there is apparently an inverse relationship between the bandwidth and the largest observed slope, letting us think that such overhead is unlike to be observed in practice. We can conclude that these results clearly emphasize the need for a characterization of the nature of memory behavior beyond the microbenchmarks' parameters.

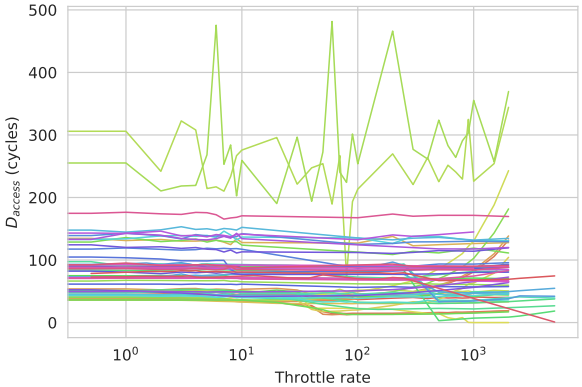
B. Qualitative metrics

The purpose of qualitative metrics is to quantify aspects on the nature of memory traffic. From our previous results, we propose the following metrics.

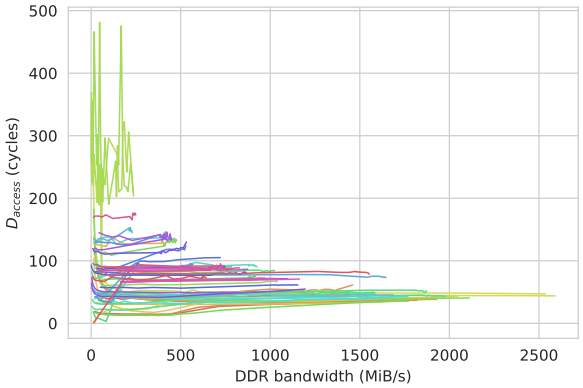
1) Read over write ratio: Read and write access requests are affected differently by the problem of interferences. On the MMDC of our platform, each type of requests have a different queue and are treated differently [17]. Moreover, they do not affect the application progress in the same way. By their semantic, read accesses access are synchronous unless they are prefetched. Thus, they are more likely to block applications progress than write accesses which are semantically



(a) Bandwidth in isolation vs. overhead (each point represents a microbenchmark instance)



(b) Throttle rate vs. average delay per access



(c) Throttle rate vs. average delay per access

Fig. 4: Summary of bandwidth characterization of 1568 microbenchmark instances. Each line and color indicates a memory consumption nature.

asynchronous. We account these differences with the ratio of read over write access requests. We measure it using the performance counters located in the DRAM controller of our platform.

2) Access type interleaving: The interleaving of access requests can have different impacts. Regarding the application progress, highly interleaved read and write accesses can (but not always) indicate more data dependencies. Another impact is for the service time of these requests by DRAM. Switching the type of access induces *data bus inversions*, which are costly in terms of DDR timings [18]. On our experimental platform, accesses causing data bus inversions are penalized by the MMDC access request scheduler [12]. Consequently, applications yielding highly interleaved traffic are more likely to suffer from unfairness during the access reordering, thus are more sensitive to interferences.

3) Access pattern entropy: The access pattern of an application defines how accesses jump from a memory location to another. If the address has b bytes, the jump between address a_1 and a_2 is $(a_2 - a_1) \bmod (2^b - 1)$. On Figure 1 read and write accesses follow two different patterns: reads are sequential (their arrows are parallel) and writes seems to be mostly random (their arrows are entangled). We want to quantify this degree of randomness. To do so, we borrow the concept of *self-information* from information theory [19]. The self-information I of a jump j in an access pattern P is a measure of the information brought by j to deduce the whole pattern. It is based on the probability of occurrence of j in P noted $p(j)$.

$$I(j) = -log(p(j)) \tag{5}$$

The idea of self-information is that events happening often do not give much information. For instance, in a sequential pattern where there is only one possible jump value, we can deduce the whole pattern examining only one element of the pattern. There is no jump value giving more information than another since they are all identical. In this case, the self-information of any element of the pattern is zero since $p(j) = 1$. We measure the complexity of the access pattern P using its Shannon entropy $H(P)$, which is the average self-

information of the elements of the sequence. It is also the number of bits required to encode the whole pattern.

$$H(P) = \mathbb{E}[I(P)] = -\sum_{j \in P} p(j) log(p(j)) \quad (6)$$

Applications with high entropy are likely to have poor locality, and consequently an increased memory consumption of system components such as DRAM rows. It is also a measure of difficulty for the speculation mechanisms to work effectively. Self information can be seen as a cost for a prefetcher to accurately predict future accesses from the past, and by extension to mitigate the impact of interferences.

4) Impact of service time degradation: The nature of the memory consumption of an application does not only affect the time taken by the memory controller to serve access requests but also how this service time impacts applications progress. Instructions causing memory accesses have a significantly longer completion time than others. Since the intensity I of the memory traffic is the number of accesses by instruction, it has a significant impact on the application's progress speed measured in *cycles per instruction (CPI)*. In fact, the CPI rate of an application can be expressed as the combination of the CPI of instructions producing memory accesses and the CPI of other instructions.

$$CPI^{iso} = I \cdot CPI^{iso}_{mem} + (1 - I) \cdot CPI^{iso}_{comp} \quad (7)$$

The value of CPI^{iso}_{mem} is highly dependent on the time T_{serv} taken by the memory controller to serve memory accesses. On average, this time depends on the nature of the traffic. For instance, an access request can be served much more quickly if the corresponding row is already loaded in the row buffer. Thus, accesses in a memory traffic exhibiting poor locality will be longer to serve *on average*, as corresponding rows are less likely to be loaded. However, for architectural reasons (pipelines, out of order execution, non-blocking caches, ...), T_{serv} is rarely exactly reflected in the value of CPI_{mem}. We express this difference using an impact factor noted τ.

$$CPI_{mem} = \tau \cdot T_{serv} \quad (8)$$

By injecting this definition of CPI_{mem} in Equation 7 we can compute τ as follows.

$$\tau = \frac{CPI - (1 - I) \cdot CPI_{comp}}{I \cdot T_{serv}} \quad (9)$$

The value of τ is rarely equal to 1. In practice it can be lesser or greater. The first case indicates that the hardware is able to compensate the DRAM service time, thanks to features like pipelining and out-of-order execution. Conversely, the second case τ indicates that the service time of the DRAM is only a part of the overall memory access delay. This can be explained by the impact of other memory system components, but also by timing anomalies [20].

We measure T_{serv} using the PMU of the MMDC. The intensity is measured using the number of accesses measured on the MMDC and the number of instructions measured on the core's PMU. For the value of CPI_{comp}, we use a conservative approach and take the highest CPI value achievable on the platform.

C. Profiling platform

Most of the qualitative metrics we just described are not measurable using hardware counters. That is why we must use some kind of simulation in order to measure them. The problem is that COTS hardware generally has two characteristics: it is rather complex and its precise behavior is not thoroughly documented. We avoid this drawback by sticking to our event based memory consumption representation: the memory consumption of an application is a stream of access requests emitted to a black box shared memory system. In our setup, the shared memory is the DRAM, hence we consider that an access is triggered on L2 cache misses. We have developed a tool to reproduce the stream of access requests yielded during an application execution. It is based on the CacheGrind platform of the VALGRIND [4] dynamic binary instrumentation framework. Except for the cache hierarchy, the target hardware is treated as a black box.

The workflow to obtain a high resolution profile is described in Figure 5. It begins with the compilation of the application (step 1). No special compilation flags are required. The binary is then executed in VALGRIND. The program execution is emulated basic block by basic block. Each basic block is decompiled in intermediate representation, then this representation is instrumented; finally, the instrumented block is compiled back to native language and executed. The cache simulation and traffic capture are performed by the instrumentation (step 2).

This method allows us to have a detailed view of the memory traffic without relying on hardware implementation details. Because this is not a cycle accurate simulation, the progression cannot be measured with time. Our answer is to measure application progress in terms of executed instructions, as it is sufficient to characterize applicative behaviors. We also offer a checkpointing feature. A checkpoint is a hook in the code that tells the profiler to mark a position in the emulated stream. We implement this feature using VALGRIND's monitor interface. Checkpoints are later converted to measure points, allowing us to gather performance metrics on portions of the captured stream.

A limitation of our approach is that we only reproduce the traffic caused *explicitly* by the application. The consequence is that the implicit traffic, generated for instance by prefetchers, is not captured by our tool. This restriction could be lifted with further development on the profiling tool. Nevertheless, this would require a deep understanding of the underlying hardware, which is not always achievable on COTS hardware. Moreover, since the hardware model we use is rather abstract, the traffic we capture for an application can be reused on hardware platforms with the same ISA and the same cache size. This would not be necessarily the case with a more concrete hardware model.

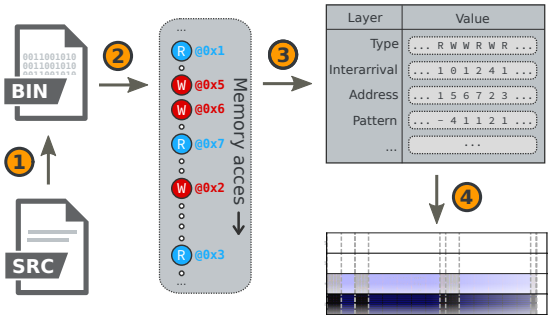


Fig. 5: Profiling workflow

D. High resolution profiles

The data gathered during the second step of the workflow illustrated in Figure 5 are organized flexibly using a *layered high resolution profile format*. Just as layers in an image format map an aspect of each of its pixels (transparency for instance), layers in our profile format map an aspect of each of its access requests. The whole raw traffic can be represented using three layers. The first layer maps the type of each access request. The second layer maps the address each access refers to. Finally, the third layer maps the number of instructions executed after each access.

The representation of the memory traffic can be easily enriched by stacking additional layers. For instance, we derive the address layer in several sublayers. The first derivation is a split of the address corresponding to the platform's L2 cache interpretation: offset, index, tag's lower half, and tag's upper half bits. The next derivation is the pattern associated with every address layers, in other words, the sequence of jumps between consecutive addresses.

The likeliness of our high resolution profiling format with an image format makes it particularly well suited for graphical representation. Three examples of profiles taken from applications of the MIBENCH and the PARSEC suites are given in Figure 6. The patterns associated to each part of the address are clearly distinct. Sequential patterns are indicated by lines, while more complex patterns tend to be represented by noise. Patterns can be complex while preserving a structure. For example, we can observe interleaved sequential patterns. This complexity can be measured using the entropy metrics defined in Section III-B. We can clearly observe distinct phases in the application's execution, which cannot be easily identifiable if they have similar performances.

The checkpointing feature offered by our profiler allows us to validate the placement of measure points according to applications phases. This permits refinement of the analysis by splitting an application in several phases exhibiting a homogeneous behavior.

E. Case study

Using our profiler with all the instances of `bodytrack` (`small`, `medium` and `large`), we split these applications in 21 phases. Moreover, our high resolution profiles allows us to distinguish three distinct behaviors we note $B1$, $B2$ and $B3$, as annotated in Figure 6a.

Figure 7 illustrates, for each behavior, respectively the bandwidth, global overhead, and the delay per access D_{access}. This study is interesting for three reasons. It confirms the relevance of our splitting, as the phases tagged with the same behavior in our profiles have similar bandwidth and global overhead. It also confirms the relevance of our qualitative metrics. Indeed $B2$ and $B3$ have the same bandwidth but they respectively suffer overheads around 100% and 20%. Finally, despite their difference of bandwidth, $B1$ and $B3$ are quite similar on the profile. We retrieve this similarity in their almost equal D_{access} values.

IV. OVERHEAD INFERENCE

In this section, we discuss the usage of our metrics, our profiler and our microbenchmarks to infer the sensitivity of an application regarding memory interference. We first present our inference methodology, then we use it to evaluate the relevance of our qualitative characterization.

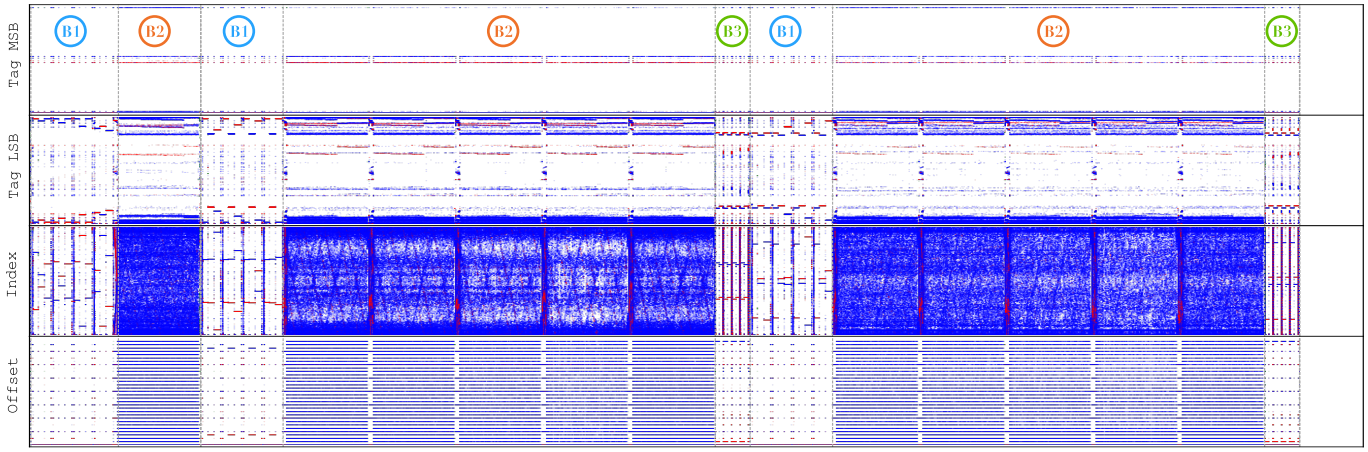
A. Regression methodology

We use a regression algorithm to predict the worst observed overhead suffered by an application given a set of metrics characterizing its behavior in isolation. We chose to use random forests [5] because it is well suited for the regression of non-linear models and it is robust against overfitting. We use the implementation of the Scikit-learn [21] machine learning library.

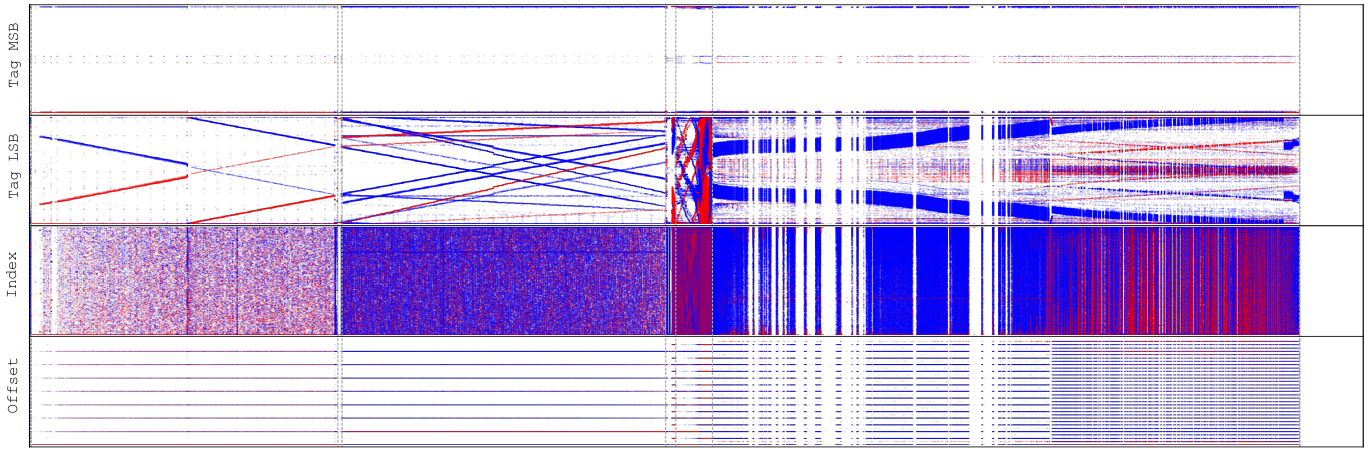
We constitute a training and a validation set to respectively fit the regressor and evaluate its precision. Each element of these sets is a measure (X, y), X being a vector of characteristics of the memory traffic in isolation, and y the worst measured overhead. In the training set, y is meant to capture the platform behavior, while in the validation set y is used to compute the prediction error. In our evaluation, training set data are measured from the microbenchmark instances presented in section II-D, while validation set data are measured from a set of applications of the MIBENCH [22] and the PARSEC [7] suites. We use the checkpointing facility provided by our profiler to split the applications' executions into phases. We do not consider the initialization phases of any application in our experiments.

B. Evaluation

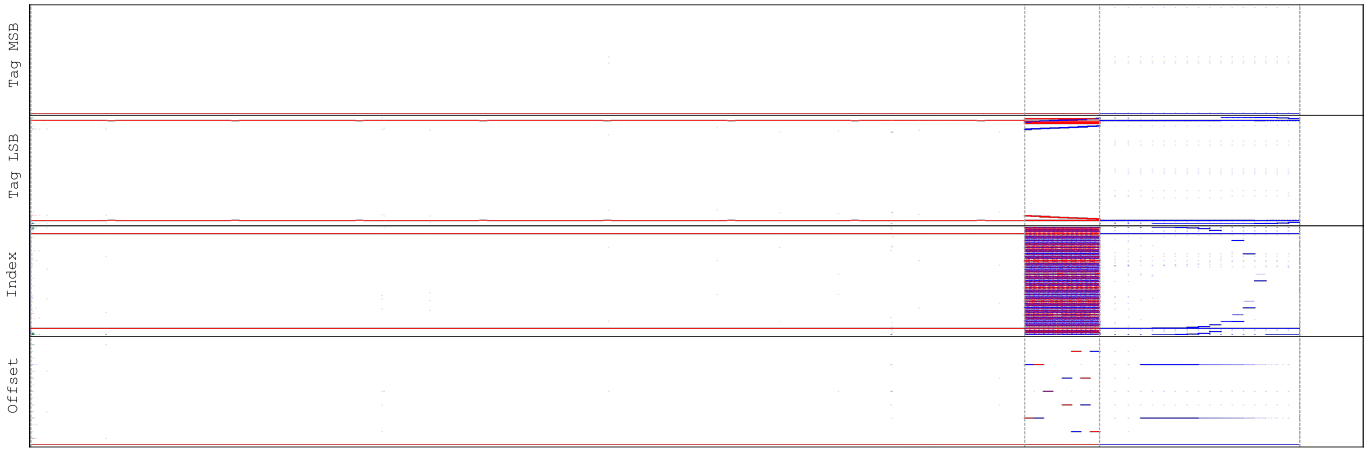
We evaluate the precision of the overhead inference, depending on which metrics are used to characterize the memory traffic in isolation. To do so, we consider combinations of three feature sets: B is the set comprising the bandwidth in isolation, Q is a set comprising qualitative features, and τ is a set comprising the τ impact factor. The Q set comprises all the qualitative features presented in Section III-B excepted τ: the read over write ratio, the interleaving rate of read and write accesses, the pattern entropy for the whole and the split address. The address is split in four parts: offset (bits $[0, 5[$), index (bits $[5, 16[$), tag's lower half (bits $[16, 24[$), and tags



(a) `bodytrack-medium`

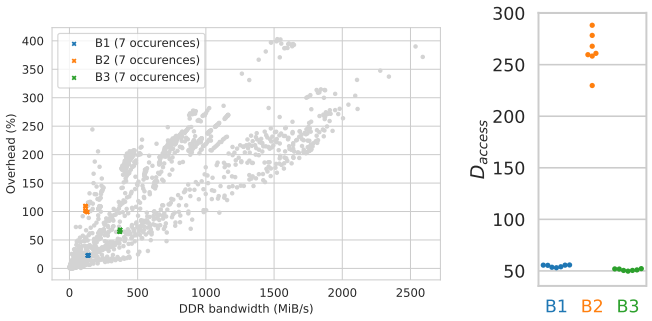


(b) `freqmine-small`



(c) `fft-large`

Fig. 6: Graphical display of the memory access stream captured for application from the MiBench and PARSEC benchmark suites. There are four stacked graphs sharing the x-axis. The x-axis accounts for the number of instructions executed. For each graph, the y-axis represents the difference between consecutive parts of the address. The address is split according to the mapping used by the last level cache of our experimental platform. For instance, the value of the ith point in the `index` layer is $(index_i - index_{i-1}) \bmod N_{index}$, where $index_i$ and N_index are respectively the value and the number of indices encoded by the index bits in the destination address of the ith access. Each color represents a type of access, red for writes and blue for reads. Finally, the horizontal dotted lines represent the checkpoints we placed in the application.



(a) (b)

Fig. 7: Sensitivity of `bodytrack` behaviors

upper half (bits $[24, 32[$). We present the results for three combinations of features: B, BQ and $BQ\tau$. B is the baseline, BQ and $BQ\tau$ are meant to evaluate respectively the relevance of qualitative metrics and the impact factor τ.

We measure the quality of prediction of the validation set with the *Mean Squared Error (MSE)* and the *Mean Absolute error (MAE)*.

$$MSE(M) = \frac{1}{|M|} \sum_{m \in M} (predicted(m) - observed(m))^2 \tag{10}$$

$$MAE(M) = \frac{1}{|M|} \sum_{m \in M} |predicted(m) - observed(m)| \tag{11}$$

The base of comparison is the worst overhead measured using the protocol described in Section II-C. The MSE gives a higher penalty to high mispredictions; hence this metric is useful to evaluate the range of mispredictions. Whereas, the MAE gives the average prediction error encountered on the dataset.

The prediction error per application and for the whole validation set is summarized in the Table III. There is a clear improvement of prediction precision from the baseline for the BQ and the $BQ\tau$ sets. This indicates the relevance of the metrics in the Q and the τ sets to characterize interference sensitivity, but also that our microbenchmarks offer a good coverage of memory consumption nature and intensity. The greatest improvements are observed for `streamcluster-small`, `canneal-small`, and `canneal-medium`. The effect of the τ impact factor is clearly more important on the MSE than on the MAE, indicating that it reduces the range of mispredictions. The gain of accuracy brought by the Q and the τ sets is observable on the Figure 8 showing the predicted and the observed values of each test case.

V. Related work

We distinguish three families of approaches to tackle the problem caused by memory interference. The first one consists in the extension of timing analysis techniques used to determine the WCET of real-time applications. The second one regroups the approaches aiming at improving the isolation of applications running on the same hardware in order to avoid interference. Finally, the third family of approaches aim at managing the effect of interference to respect timing requirements.

TABLE III: Mean squared and absolute prediction error (MSE and MAE) per application and for the whole validation set

Prediction error metric	MAE			MSE		
Application	B	BQ	$BQ\tau$	B	BQ	$BQ\tau$
adpcm-c-small	32.4	2.6	16.6	1052.9	6.8	276.5
adpcm-c-large	40.5	5.8	13.2	1638.2	33.7	174.0
adpcm-d-small	65.3	6.1	19.8	4257.7	37.8	392.5
adpcm-d-large	63.0	6.0	19.7	3970.5	36.6	388.6
blackscholes-small	10.3	8.2	8.4	106.2	66.9	70.9
blackscholes-medium	21.9	4.3	4.6	479.8	18.4	21.4
blackscholes-large	28.4	1.2	4.5	804.1	1.5	19.9
bodytrack-small	47.9	25.8	17.6	3103.3	1424.3	581.6
bodytrack-medium	34.3	24.5	17.2	1546.4	1225.2	504.9
bodytrack-large	25.9	22.5	15.2	837.1	1060.3	435.7
canneal-small	141.0	16.5	25.9	19872.2	271.5	669.3
canneal-medium	158.0	22.2	32.8	24966.0	493.8	1073.9
fft-small	22.7	15.0	19.5	797.7	651.5	704.1
fft-medium	29.0	18.0	16.8	1620.4	492.2	421.6
fft-large	20.0	11.3	9.4	769.3	311.3	216.6
freqmine-small	34.2	32.6	28.0	2056.1	1818.3	1344.1
freqmine-medium	33.3	21.7	23.7	2221.9	923.2	1016.1
freqmine-large	52.4	30.8	26.2	5241.8	2548.8	1416.9
patricia-small	5.9	15.3	5.6	35.0	234.2	31.4
patricia-large	6.8	14.6	9.7	45.7	212.0	94.9
qsort-large	52.3	68.2	43.1	2734.3	4644.5	1861.1
rijndael-dec-small	6.3	2.7	2.8	39.9	7.3	8.1
rijndael-dec-large	7.1	1.2	2.7	51.1	1.5	7.4
rijndael-enc-small	2.5	6.8	3.3	6.0	46.8	10.7
rijndael-enc-large	26.7	5.3	0.6	713.5	27.7	0.4
sha-small	7.9	0.2	11.4	61.7	0.0	130.4
sha-large	11.0	9.4	19.3	120.0	88.7	370.9
streamcluster-small	95.7	43.9	40.5	11056.9	2332.2	2032.7
Validation set	37.4	21.7	18.8	2667.0	1093.7	681.6
Error loss from B	-	**42.0%**	**49.9%**	-	**59.0%**	**74.4%**

A. Timing analysis

Timing analysis is the step of real-time system design where the WCET of applications is determined. Interferences in multi-core complicate timing analysis, as the WCET of an application does not only depend on itself but on the behavior of its co-runners.

Static timing analysis as done on single-core systems [23] has proven to be hard to transpose on multi-core systems. A detailed survey on the state of the art of multi-core timing analysis has been done by Maiza et al. [1]. The application of static timing analysis requires a deep knowledge of the underlying hardware platform, which is not always achievable with COTS hardware. For instance, Kim et al. [24] present a timing analysis that computes a bound of DRAM response time degradation based on information provided by JEDEC [18] documentation. While permitting to derive safe bounds, this kind of work currently suffers several limitations. First, the derived overheads are usually very pessimistic. Second, some assumptions are usually made regarding the compositionality [20] of the platform. In other words, the ability to compute the overall delay from the delays suffered by individual components of the application. On our platform, this result cannot be applied as it is invalidated by the behavior of the MMDC. Oehlert et al. [25] use static analysis to extract event arrival functions giving the number of access

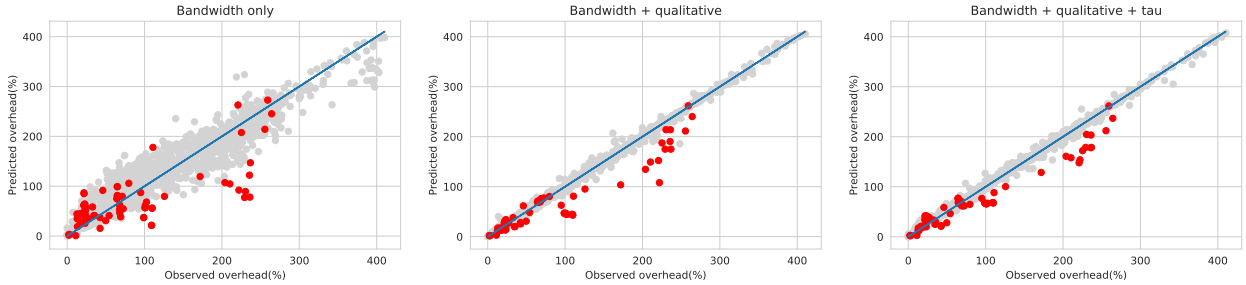


Fig. 8: Predicted vs. observed values. Gray and red points belong respectively to the training and the validation set.

triggered for a time window. Such technique can be used to determine a safe approximation of the memory consumption of an application for every execution paths. However, our results show that a purely quantitative characterization of the memory trafic is imprecise.

Many studies use dynamic methods to evaluate the effect of interference, notably the ones conducted by Bin et al. [26], Radojkovic et al. [27], Nowosch et al. [28], Fernandez et al. [29], and Zhuravlev et al. [30]. Nevertheless, contrary to our work they characterize applications in terms of consumption instead of behavior. Griffin et al. [31] tackle the same problem as ours. They use Deep Neural Networks to infer the overehead suffered by applications from metrics gathered using hardware performance counters. Our approach differ by the choice of the metrics used to characterize consumption behaviour, and it is less dependent on the availability of good quality counters on the platform. Black-Schaffer et al. [32] and Mars et al. [33] propose an interference sensitivity characterization to infer the performance degradation caused by co-locations in rack-scale computers, while Funston et al. [34] propose an automatic characterization generation for that purpose. Unlike these approaches, our characterization relies exclusively on the behavior of the application in isolation, and load injection is only required during the training set constitution for a given platform. The aggressivity of loads used during the constitution of a training set is an important aspect of measure based approaches. Iorga et al. [35] propose an autotuner to determine parameters for load applications to generate more interference. Such tools can be beneficial to our approach.

B. Isolation

The effects of interferences can be mitigated, or even prevented, by avoiding collisions on shared hardware. *Spatial isolation* is achieved by making shared resources private, whereas *temporal isolation* is achieved when two cores cannot access a shared resource simultaneously. The achievable level of isolation is highly dependent on the hardware platform. Time predictable architectures such as PRET [36], T-CREST [37], or MERASA [38] provide mechanisms allowing these two types of isolation. However, since they are not widely available, they are not in the scope of this article.

Spatial isolation can be achieved with hardware support, or software methods like page coloring [39]. Page coloring can be used to partition resources as diverse as physically indexed caches [40], TLB entries [41], DDR channels [42], or DRAM banks [43]. Nevertheless, it requires operating system support and knowledge of the mapping of physical address to components location. PALLOC [43] address these two issues for the Linux kernel with a color-aware page allocator and an experimental methodology to determine the mapping used by the target DDR controller. Further operating system support for coloring is proposed by Ward et al. [44]. They study the management of cache colors as a shared resource, and propose a scheduling analysis using a single core equivalence. Spatial isolation is relatively easy to provide. However it induces an increased intra-core pressure on partitioned resources resulting in performance degradation. The tradeoff between the gain of determinism and performance degradation is studied extensively by Kim et al. [45]. These approaches allow us to leverage spatial partitioning without depending on hardware support. Although our approach is applicable without using spatial partitioning, doing so change the definition of what constitutes the memory system. Thus, it may requires some adjustements regarding quantitative characterization. The application of our appoach with different partitioning setups is a promising research direction we plan to explore in the future.

Temporal isolation completely negates interferences by scheduling the access to shared resources. Unlike spatial isolation, it is very difficult to achieve efficiently without specific hardware. Fischer et al. [46] provides temporal isolation to critical tasks in mixed criticality systems, by disabling all but one core when a time critical application executes, which results in important resource underutilization. One way to achieve a finer grain temporal isolation is to use a phased execution model such as the superblock model [47] [48] or PREM [49]. In such models, shared memory can only be accessed during explicit data acquisition and restitution phases, while only local resources are used in computation phases. This separation allows the system to schedule data phases sequentially and computation phases in parallel. Although they improve platform utilization, such models require rewriting applications to comply to them, making this approach incompatible with legacy software. Systems providing fine-grained temporal isolation have been proposed by Jean et al. [50] on a PowerPC platform, and by Perret et al. [51] on the

Kalray MPPA-256 platform, although these implementations rely on platform-specific features. Our approach could be used to improve the performance of this kind of approach by determining the sensitive phases for which strict temporal isolation should be enforced and insensitive one that do not require this level of protection against interference.

C. Interference regulation

Regulation systems [52] can be seen as a relaxed form of temporal isolation. Instead of avoiding interference at all cost, a regulation system manages resources so that time-critical applications fulfill their deadline requirements. This kind of approach is well suited for *mixed criticality systems*, where they can be used to protect real-time applications from the interference caused by non-critical applications.

Kritikakou et al. propose a *distributed WCET controller* [53] [54] to that end. In this approach, the code of real-time applications is instrumented with observation points in charge of tracking application progress. If an observation point detects an unacceptable slowdown in the application progress, it sends a request to a distributed WCET controller to suspend non-critical tasks. The main difficulty to apply this method is the placement of observation points, although we believe our approach could provide some assistance for that task. Indeed, our high resolution profile format allows to clearly distinguish applications phases, hence to identify zone of interests in applications code. Moreover, the clearly distinctive visual pattern opens the possibility to use computer vision techniques to automate applications splitting.

An alternative approach is proposed by Blin et al. [3] to keep under a specified threshold the slowdown caused by best efforts application to a real-time one. In this approach, an interference model of the hardware platform is built using measures. The model associates a local and a global bandwidth to the worst observed overhead suffered by an application consuming the local bandwidth in isolation when the global bandwidth is observed. Knowing the consumption of the real-time application, a run-time control system periodically use the interference model to infer the delay the application suffers, and suspend best effort tasks when the cumulated overhead reaches a threshold specified by the system designer. Our approach can be directly beneficial to this approach as it consists in generating accurate interference model. However, some extensions are required to predict the overhead suffered under a specific system load.

Memguard [2] is a bandwidth regulator. Each core has a memory access budget. Hardware performance counters are used to track the number of memory access made by each core. When an application spends all its access budget, an interruption is raised and the task is suspended until the next regulation period. The level of interference is managed by keeping the overall bandwidth below a threshold, resulting in hardware resource underutilization. Our approach could be used to account the sensitivity of applications in a workload and adjust the guaranteed bandwidth thresholds accordingly in order to improve platform utilization.

VI. Conclusion and future work

In this article, we have presented an experimental approach to characterize relevant aspects of the memory consumption to their sensitivity to memory interferences. We have introduced a set of microbenchmarks allowing us to reproduce a wide range of memory consumptions, both in nature and intensity, at a sufficiently large scale to use machine learning techniques. We have put in evidence the lack of precision of a purely quantitative characterization of memory consumption. In response, we have defined qualitative metrics quantifying the type of accesses, their interleaving, locality, or their impact on application progress. We have developed a tool to measure those aspects that also capture high resolution profiles allowing us to clearly identify distinct consumption behaviors in an application. Finally, we have used our microbenchmarks and our characterization to infer the overhead suffered by applications of the MiBench and the PARSEC suites. Results show a significant improvement of quality prediction compared to a purely quantitative characterization, showing the relevance of our metrics and the representativity of our microbenchmarks.

In the future, we will explore several directions to improve this work. The first direction we want to explore is to use our approach to evaluate more hardware platforms. The second one is to build a set of conservative predictor that does not allow underestimations. This can involve the development of specific inference algorithms or new interference models incorporating our characterization. The third direction is to widen the scope of consumption behavior covered by our microbenchmarks and to find new relevant metrics. Finally, the last direction we want to explore is the automation of measure points placement, using computer vision techniques and code analysis.

Acknowledgements

We would like to offer special thanks to our colleagues Redha Gouicem and Damien Carver for their precious help in the improvement of this manuscript.

References

[1] C. Maiza, H. Rihani, J. M. Rivas, J. Goossens, S. Altmeyer, and R. I. Davis, "A survey of timing verification techniques for multi-core real-time systems," 2018.

[2] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha, "Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms," in *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2013 IEEE 19th*. IEEE, 2013, pp. 55–64.

[3] A. Blin, C. Courtaud, J. Sopena, J. Lawall, and G. Muller, "Maximizing parallelism without exploding deadlines in a mixed criticality embedded system," in *Real-Time Systems (ECRTS), 2016 28th Euromicro Conference on*. IEEE, 2016, pp. 109–119.

[4] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," in *ACM Sigplan notices*, vol. 42, no. 6. ACM, 2007, pp. 89–100.

[5] L. Breiman, "Random forests," *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.

[6] Sabre board for smart devices based on the i.mx 6quad applications processors. [Online]. Available: https://www.nxp.com/support/developer-resources/evaluation-and-development-boards/sabre-development-system/sabre-board-for-smart-devices-based-on-the-i.mx-6quad-applications-processors:RD-IMX6Q-SABRE

[7] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*. ACM, 2008, pp. 72–81.

[8] J. D. McCalpin, "Memory bandwidth and machine balance in current high performance computers," *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pp. 19–25, Dec. 1995.

[9] *i.MX 6Dual/6Quad Applications Processor Reference Manual*, p.204-205.

[10] *PL310 Cache Controller Technical Reference Manual*, p.327-328.

[11] P. K. Valsan, H. Yun, and F. Farshchi, "Taming non-blocking caches to improve isolation in multicore real-time systems," in *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2016, pp. 1–12.

[12] *i.MX 6Dual/6Quad Applications Processor Reference Manual*, p.3852.

[13] The yocto project. [Online]. Available: https://www.yoctoproject.org

[14] Technical details of preempt_rt patch. [Online]. Available: https://wiki.linuxfoundation.org/realtime/documentation/technical_details/start

[15] J. M. Calandrino, H. Leontyev, A. Block, U. C. Devi, and J. H. Anderson, "Litmusˆ rt: A testbed for empirically comparing real-time multiprocessor schedulers," in *2006 27th IEEE International Real-Time Systems Symposium (RTSS'06)*. IEEE, 2006, pp. 111–126.

[16] "Scheduling - rt throttling," https://wiki.linuxfoundation.org/realtime/documentation/technical_basics/sched_rt_throttling.

[17] *i.MX 6Dual/6Quad Applications Processor Reference Manual*, p.3834-3835.

[18] JEDEC, "Specification, ddr3 sdram," 2010.

[19] C. E. Shannon, "A mathematical theory of communication," *Bell system technical journal*, vol. 27, no. 3, pp. 379–423, 1948.

[20] S. Hahn, J. Reineke, and R. Wilhelm, "Towards compositionality in execution time analysis: definition and challenges," *ACM SIGBED Review*, vol. 12, no. 1, pp. 28–36, 2015.

[21] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[22] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "Mibench: A free, commercially representative embedded benchmark suite," in *Proceedings of the fourth annual IEEE international workshop on workload characterization. WWC-4 (Cat. No. 01EX538)*. IEEE, 2001, pp. 3–14.

[23] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra *et al.*, "The worst-case execution-time problem—overview of methods and survey of tools," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 7, no. 3, p. 36, 2008.

[24] H. Kim, D. De Niz, B. Andersson, M. Klein, O. Mutlu, and R. Rajkumar, "Bounding memory interference delay in cots-based multi-core systems," in *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2014 IEEE 20th*. IEEE, 2014, pp. 145–154.

[25] D. Oehlert, S. Saidi, and H. Falk, "Compiler-based extraction of event arrival functions for real-time systems analysis," in *30th Euromicro Conference on Real-Time Systems (ECRTS 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

[26] J. Bin, S. Girbal, D. G. Pérez, A. Grasset, and A. Merigot, "Studying co-running avionic real-time applications on multi-core cots architectures," in *Embedded Real Time Software and Systems conference*, vol. 15, 2014.

[27] P. Radojković, S. Girbal, A. Grasset, E. Quiñones, S. Yehia, and F. J. Cazorla, "On the evaluation of the impact of shared resources in multithreaded cots processors in time-critical environments," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 8, no. 4, p. 34, 2012.

[28] J. Nowotsch and M. Paulitsch, "Leveraging multi-core computing architectures in avionics," in *2012 Ninth European Dependable Computing Conference*, May 2012, pp. 132–143.

[29] M. Fernández, R. Gioiosa, E. Quiñones, L. Fossati, M. Zulianello, and F. J. Cazorla, "Assessing the suitability of the ngmp multi-core processor in the space domain," in *Proceedings of the tenth ACM international conference on Embedded software*. ACM, 2012, pp. 175–184.

[30] S. Zhuravlev, S. Blagodurov, and A. Fedorova, "Addressing shared resource contention in multicore processors via scheduling," in *ACM Sigplan Notices*, vol. 45, no. 3. ACM, 2010, pp. 129–142.

[31] D. Griffin, B. Lesage, I. Bate, F. Soboczenski, and R. I. Davis, "Forecast-based interference: modelling multicore interference from observable factors," in *Proceedings of the 25th International Conference on Real-Time Networks and Systems*. ACM, 2017, pp. 198–207.

[32] D. Black-Schaffer, N. Nikoleris, E. Hagersten, and D. Eklov, "Bandwidth bandit: Quantitative characterization of memory contention," in *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE Computer Society, 2013, pp. 1–10.

[33] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa, "Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations," in *Proceedings of the 44th annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2011, pp. 248–259.

[34] J. Funston, M. Lorrillere, A. Fedorova, B. Lepers, D. Vengerov, J.-P. Lozi, and V. Quema, "Placement of virtual containers on NUMA systems: A practical and comprehensive model," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. Boston, MA: USENIX Association, 2018, pp. 281–294. [Online]. Available: https://www.usenix.org/conference/atc18/presentation/funston

[35] D. Iorga, T. Sorensen, and A. F. Donaldson, "Do your cores play nicely? a portable framework for multi-core interference tuning and analysis," *arXiv preprint arXiv:1809.05197*, 2018.

[36] I. Liu, J. Reineke, and E. A. Lee, "A pret architecture supporting concurrent programs with composable timing properties," in *2010 Conference Record of the Forty Fourth Asilomar Conference on Signals, Systems and Computers*, Nov 2010, pp. 2111–2115.

[37] M. Schoeberl, S. Abbaspour, B. Akesson, N. Audsley, R. Capasso, J. Garside, K. Goossens, S. Goossens, S. Hansen, R. Heckmann *et al.*, "T-crest: Time-predictable multi-core architecture for embedded systems," *Journal of Systems Architecture*, vol. 61, no. 9, pp. 449–471, 2015.

[38] T. Ungerer, F. Cazorla, P. Sainrat, G. Bernat, Z. Petrov, C. Rochange, E. Quinones, M. Gerdes, M. Paolieri, J. Wolf *et al.*, "Merasa: Multicore execution of hard real-time applications supporting analyzability," *IEEE Micro*, vol. 30, no. 5, pp. 66–75, 2010.

[39] R. E. Kessler and M. D. Hill, "Page placement algorithms for large real-indexed caches," *ACM Transactions on Computer Systems (TOCS)*, vol. 10, no. 4, pp. 338–359, 1992.

[40] R. Mancuso, R. Dudko, E. Betti, M. Cesati, M. Caccamo, and R. Pellizzoni, "Real-time cache management framework for multi-core architectures," in *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2013, pp. 45–54.

[41] S. A. Panchamukhi and F. Mueller, "Providing task isolation via TLB coloring," in *21st IEEE Real-Time and Embedded Technology and Applications Symposium*, 2015, pp. 3–13.

[42] S. P. Muralidhara, L. Subramanian, O. Mutlu, M. Kandemir, and T. Moscibroda, "Reducing Memory Interference in Multicore Systems via Application-aware Memory Channel Partitioning," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-44. New York, NY, USA: ACM, 2011, pp. 374–385, event-place: Porto Alegre, Brazil. [Online]. Available: http://doi.acm.org/10.1145/2155620.2155664

[43] H. Yun, R. Mancuso, Z.-P. Wu, and R. Pellizzoni, "Palloc: Dram bank-aware memory allocator for performance isolation on multicore platforms," in *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2014 IEEE 20th*. IEEE, 2014, pp. 155–166.

[44] B. C. Ward, J. L. Herman, C. J. Kenna, and J. H. Anderson, "Outstanding paper award: Making shared caches more predictable on multicore platforms," in *2013 25th Euromicro Conference on Real-Time Systems*. IEEE, 2013, pp. 157–167.

[45] N. Kim, B. C. Ward, M. Chisholm, J. H. Anderson, and F. D. Smith, "Attacking the one-out-of-m multicore problem by combining hardware management with mixed-criticality provisioning," *Real-Time Systems*, vol. 53, no. 5, pp. 709–759, 2017.

[46] S. Fisher, "Certifying applications in a multi-core environment: The world's first multi-core certification to sil 4," *SYSGO white paper*, 2013.

[47] G. Durrieu, M. Faugère, S. Girbal, D. Gracia Pérez, C. Pagetti, and W. Puffitsch, "Predictable flight management system implementation on a multicore processor," in *Embedded Real Time Software (ERTS'14)*, 2014.

[48] A. Schranzhofer, R. Pellizzoni, J.-J. Chen, L. Thiele, and M. Caccamo, "Timing analysis for resource access interference on adaptive resource

arbiters," in *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*. IEEE, 2011, pp. 213–222.

[49] R. Pellizzoni, E. Betti, S. Bak, G. Yao, J. Criswell, M. Caccamo, and R. Kegley, "A predictable execution model for cots-based embedded systems," in *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*. IEEE, 2011, pp. 269–279.

[50] X. Jean, "Hypervisor control of cots multicores processors in order to enforce determinism for future avionics equipment," Ph.D. dissertation, PhD Thesis, Telecom ParisTech, 2015.

[51] Q. Perret, P. Maurere, E. Noulard, C. Pagetti, P. Sainrat, and B. Triquet, "Temporal isolation of hard real-time applications on many-core processors," in *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, April 2016, pp. 1–11.

[52] S. Girbal, X. Jean, J. Le Rhun, D. G. Pérez, and M. Gatti, "Deterministic platform software for hard real-time systems using multi-core cots," in *Digital Avionics Systems Conference (DASC), 2015 IEEE/AIAA 34th*. IEEE, 2015, pp. 8D4–1.

[53] A. Kritikakou, C. Pagetti, O. Baldellon, M. Roy, and C. Rochange, "Run-time control to increase task parallelism in mixed-critical systems," in *2014 26th Euromicro Conference on Real-Time Systems*. IEEE, 2014, pp. 119–128.

[54] A. Kritikakou, C. Rochange, M. Faugère, C. Pagetti, M. Roy, S. Girbal, and D. G. Pérez, "Distributed run-time wcet controller for concurrent critical tasks in mixed-critical systems," in *Proceedings of the 22nd International Conference on Real-Time Networks and Systems*. ACM, 2014, p. 139.